Lesson 7: Designing for a Managed-Memory World

Brad Abrams
Lead Program Manager

CLR Base Class Libraries Team

April 2004

Designing for a Managed Memory World

- After successfully completing this lesson, you will be able to:
  - Understand the effect garbage collection has on library design
  - Effectively design classes for this environment

Garbage Collection

- How does it affect API design?
  - Of course, no leaks or stray pointers
  - No memory ownership issues
  - Enables functional programming model

Garbage Collection (2)

- Local vs. global view
  - Possible to build faster domain-specific memory and resource management, but gains are lost in compound solutions
  - For example, no common string type
- Issue: Non-deterministic lifetimes

Resource Management

- The garbage collector (GC) does an excellent job managing "managed" memory
- GC doesn't manage external resources (database connections, HWnds, etc.)
- Generational mark-and-sweep garbage collection means non-deterministic finalization
  - Exact time of finalization is unspecified
  - Order of finalization is unspecified
  - Thread is unspecified

Resource Management (2)

- If you are encapsulating external resources:
  - Add a finalizer (C# destructor) to guarantee the resource will eventually be freed
  - Provide developers an explicit way to free external resources
  - Formalized in the IDisposable interface:
    - Signals to users they need to explicitly Dispose of instances
    - Enables C# `using` support

Destructors

- Object.Finalize is not accessible in C#
- VERY different than C++ destructors

```
public class Resource
{
    ~Resource() {
        ...
    }
}
```

```
public class Resource
{
    protected override void Finalize() {
        try {
            ...
        }
        finally {
            base.Finalize();
        }
    }
}
```

Destructors (2)

- Only implement Finalize on objects that need finalization
  - Finalization is only appropriate for clean up of unmanaged resources
  - Keeps objects alive an order of magnitude longer
- Free any external resources you own in your Finalize method
- Do not throw exceptions in finalizers
  - The rest of your finalizer will not run

Destructors (3)

- Do not block or wait in finalizers
  - All finalization for that process could be stopped
- Only release resources that are held onto by this object, and finalizer should not reference any other objects
- Will be called on one or more different threads

Dispose Pattern

- Implement the dispose pattern whenever you have a finalizer
  - Gives developers explicit control
- Free any disposable resources your type owns in the `Dispose()` method
  - Not just the external resources
  - Propagate calls to `Dispose()` through containment hierarchies

Dispose Pattern (2)

- Suppress finalization once `Dispose()` has been called
- `Dispose()` should be callable multiple times without throwing an exception
  - The method will do nothing after the first call
  - After `Dispose()` is called other methods on the class can throw ObjectDisposedException or you can re-create the internal state of the class on the fly

Dispose Pattern (3)

- Do not assume that `Dispose()` will be called
  - For unmanaged clean up, have a finalizer as well
- Provide a `Close()` that calls `Dispose()` method if 'close' is a preferred term for your type
- Do call your base class's `Dispose()` method if it implements IDisposable
- Throw an `ObjectDisposedException` on operations of a disposed type
  - On-demand re-creation is doable, but complex
- Implement the dispose pattern on base types that commonly have subtypes that hold onto resources

Implementing IDisposable

```
public class Resource: IDisposable {
   private bool disposed = false;
   pubic int GetValue () {
      if (disposed) throw new ObjectDisposedException();
      // do work
   }
   public void Dispose() {
      Dispose(true);
      GC.SuppressFinalize(this);
   }
   protected virtual void Dispose(bool disposing) {
      if (disposing) {
         // Dispose dependent objects
         disposed = true;
      }
      // Free unmanaged resources
   }
   ~Resource() {
      Dispose(false);
   }
}
```

Using Statement

- Acquire, execute, release pattern
- Works with any IDisposable object
  - Data access classes, streams, text readers and writers, network classes, etc.

```
using (Resource res = new Resource()) {
    res.DoWork();
}
```

```
Resource res = new Resource(...);
try {
    res.DoWork();
}
finally {
    if (res != null)
        ((IDisposable)res).Dispose();
}
```

Using Statement (2)

- Can you find the "bug" in this code?
- Will input and output always be closed?

```
static void Copy(string sourceName, string destName) {
   Stream input = File.OpenRead(sourceName);
   Stream output = File.Create(destName);
   byte[] b = new byte[65536];
   int n;
   while ((n = input.Read(b, 0, b.Length)) != 0) {
      output.Write(b, 0, n);
   }
   output.Close();
   input.Close();
}
```

Using Statement (3)

```
static void Copy(string sourceName, string destName) {
   Stream input = File.OpenRead(sourceName);
   try {
      Stream output = File.Create(destName);
      try {
         byte[] b = new byte[65536];
         int n;
         while ((n = input.Read(b, 0, b.Length)) != 0) {
            output.Write(b, 0, n);
         }
      }
      finally {
         output.Close();
      }
   }
   finally {
      input.Close();
   }
}
```

Using Statement (4)

- Code is correct and much more readable with using statements
- Types should implement IDisposable to take advantage of this support
- Provide a well-known named method that maps to privately implement Dispose method if appropriate

```
static void Copy(string sourceName, string destName) {
   using (Stream input = File.OpenRead(sourceName))
   using (Stream output = File.Create(destName)) {
      byte[] b = new byte[65536];
      int n;
      while ((n = input.Read(b, 0, b.Length)) != 0) {
         output.Write(b, 0, n);
      }
   }
}
```

Resource Management
"Whidbey Feature": MemoryPressure

- GC.AddMemoryPressure (int pressure)
  - Useful when you have a disproportionate ratio of managed-to-unmanaged resources
  - Garbage collection (GC) alters its strategy, to increase the number of collections performed
  - GC.RemoveMemoryPressure when your object is freed, to allow the GC to return to its standard strategy

Resource Management
"Whidbey" Feature: MemoryPressure

```
class Bitmap {
        private long _size;
        Bitmap (string path ) {
            _size = new FileInfo(path).Length;
            GC.AddMemoryPressure(_size);
            // other work
        }
        ~Bitmap() {
            GC.RemoveMemoryPressure(_size);
            // other work
        }
}
```

Resource Management
"Whidbey" Feature: HandleCollector

- HandleCollector keeps track of a limited number of handles
  - Typically, unmanaged resource handles: HDCs, HWnds, etc.
- When you allocate a new handle, call Add
- When you are freeing, call Remove
- As you add to the collector, it may perform a GC.Collect, to free existing handles, based on the current count and the number of resources available

```
HandleCollector(string name, int initialThreshold,
                int maximumThreshold);
```

name: Allows you to track each handle type separately, if needed

initialThreshold: The point at which collections should begin being performed

maximumThreshold: The point at which collections MUST be performed—this should be set to the maximum number of available handles

Resource Management
"Whidbey Feature": HandleCollector

```
static readonly HandleCollector GdiHandleType =
        new HandleCollector( "GdiHandles", 10, 50);

static IntPtr CreateSolidBrush() {
        IntPtr temp = CreateSolidBrushImpl(…);
        GdiHandleType.Add();
        return temp;
}

internal static void DeleteObject(IntPtr handle) {
        DeleteObjectImpl(handle);
        GdiHandleType.Remove();
}
```

Exercise: What's Wrong with This Type?

```
public class File : IDisposable {
    private IntPtr fileHandle;

    public File(string path) {
        fileHandle = OpenFile (path);
    }

    public void Dispose () {
        CloseFile (fileHandle);
    }
}
```

Exercise: What's Wrong with This Type?

```
public class NetworkCache : IDisposable {
    private File localFile;
    private IntPtr socketHandle;

    public NetworkCache (string tmpPath,
                         string url) {
        localFile = new File (tmpPath);
        socketHandle = OpenSocket (url);
    }
    public void Dispose () {
        localFile.Dispose();
        CloseSocket (socketHandle);
    }
    ~NetworkCache () {
        localFile.Dispose();
        CloseSocket (socketHandle);
    }
}
```

Lesson 7 Summary

- GC does a great job at handling managed memory
  - Not designed for other resources
- Use Finalizers and the dispose pattern to manage external resources
- Finalizers—free resources you own
- Dispose—propagate through the containment hierarchy
- Take advantage of the using statement